The Virtual Data Grid: An Infrastructure to Support Distributed Data-Centric Applications

Jon Weissman and Colin Gan
Army High Performance Computing and Research Center
Department of Computer Science, University of Minnesota

1.0 Introduction

Distributed data-centric applications are an increasingly important class of applications in the scientific and military domain. Often times, the data of interest is large, and distributed across different remote storage locations and data delivery networks. In the digitally enhanced battlefield of the future, data will be generated from sensor grids, a variety of national assets, and real-time simulations running on high-performance computers. Collectively, these sources produce continuous data of mixed traffic classes that are intermittently available and offer highly variable bandwidth both within their internal networks and to outside connections. Clients may wish to access these data sources from a variety of hand-held or wearable devices including low-power wireless devices in the field. Furthermore, different clients may expect to receive different views of the data depending on their context and their role. The future digitally enhanced battlefield will require such customized and pro-active delivery of critical data products to military personnell in combat and to high-performance data-intensive applications running on their behalf, e.g. data mining. Data from multiple distributed sources must be delivered reliably, often in real-time, while masking the heterogeneity of storage systems and data formats.

In this design paper, we introduce an infrastructure called the Virtual Data Grid (VDG) that provides reliable, high performance, data-delivery services for large-scale scientific applications and end-users that require access to distributed data. The VDG presents a persistent, continuously available data network that masks the intermittency, heterogeneity, and distribution of the data sources that feed it. The VDG infrastructure utilizes metacomputing technology to support data-centric applications by providing seamless management and delivery of heterogeneous data sources for Army applications. We are applying the VDG concept to support next-generation synthetic battlefield simulation as part of the AHPCRC. We will describe a motivating application, the current status of the VDG system architecture, and the research challenges we are addressing in the area of data scheduling.

2.0 A Motivating Scenario

Military personnel may want to access a large amount of data distributed across different remote storage locations in a battlefield scenario. This is critical in the proposed new Network Centric Combat model by the Army encompassing different data sources, including:

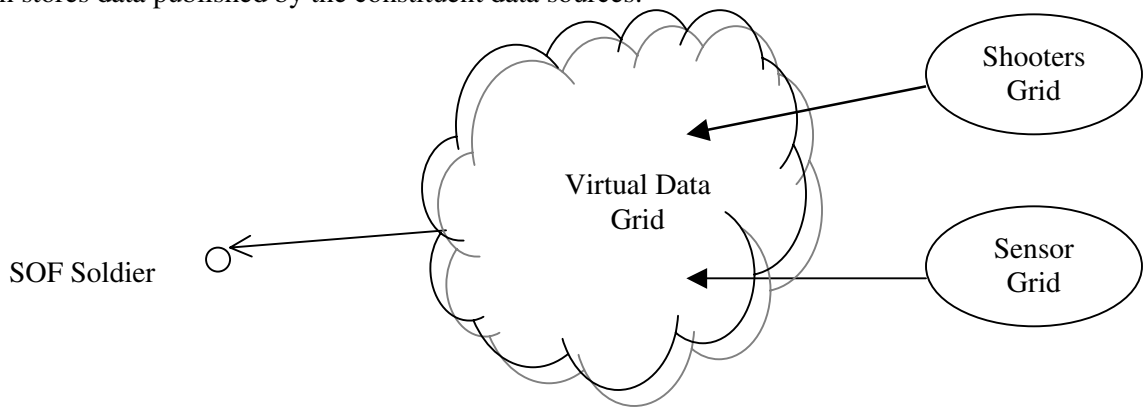
a) Information Grid. This provides the infrastructure for Network Centric Computing and Communications. It provides the means to receive, process, transport, store and protect

information for the Joint and combined forces. Its embedded capabilities for Information Assurance prevent intrusive attack and assure commanders that their information will be valid. It provides the necessary infrastructure to permit the plug and play of the sensors and shooters. It will exist in space in both low and high-earth orbit, in the air at all altitudes, on land and under the sea. It is a physical permanent grid.

b) Sensor Grid. This is composed of air, sea, ground, space and cyberspace-based sensors, and includes dedicated sensors, sensors based on weapons platforms, and sensors employed by individual soldiers and embedded logistics sensors. It provides the Joint force with a high degree of awareness of friendly forces, enemy forces and the environment across the Joint Battlespace. It is a transient grid. The sensors are physical and when tasked to produce information about a target they are interrelated, the grid exists for the task only and is amended for every mission.

c) Shooter Grid. This enables the Joint Warfighter to plan and execute operations in a manner achieving an overwhelming effect at precise places and time. It is like the sensor grid where the piece parts are physical but the grid is only virtual. The shooters are tasked to create the necessary effect on the battlefield then dynamically retasked when nececssary.

Military personnel may wish to access different data objects from Information, Sensor or Shooter Grids from a variety of hand-held or wearable devices including low-power wireless devices in the field. Since these various Grids may become unavailable and in the worst case destroyed by enemies, a persistent snapshot of the data must be available. This is maintained within the VDG, which stores data published by the constituent data sources.

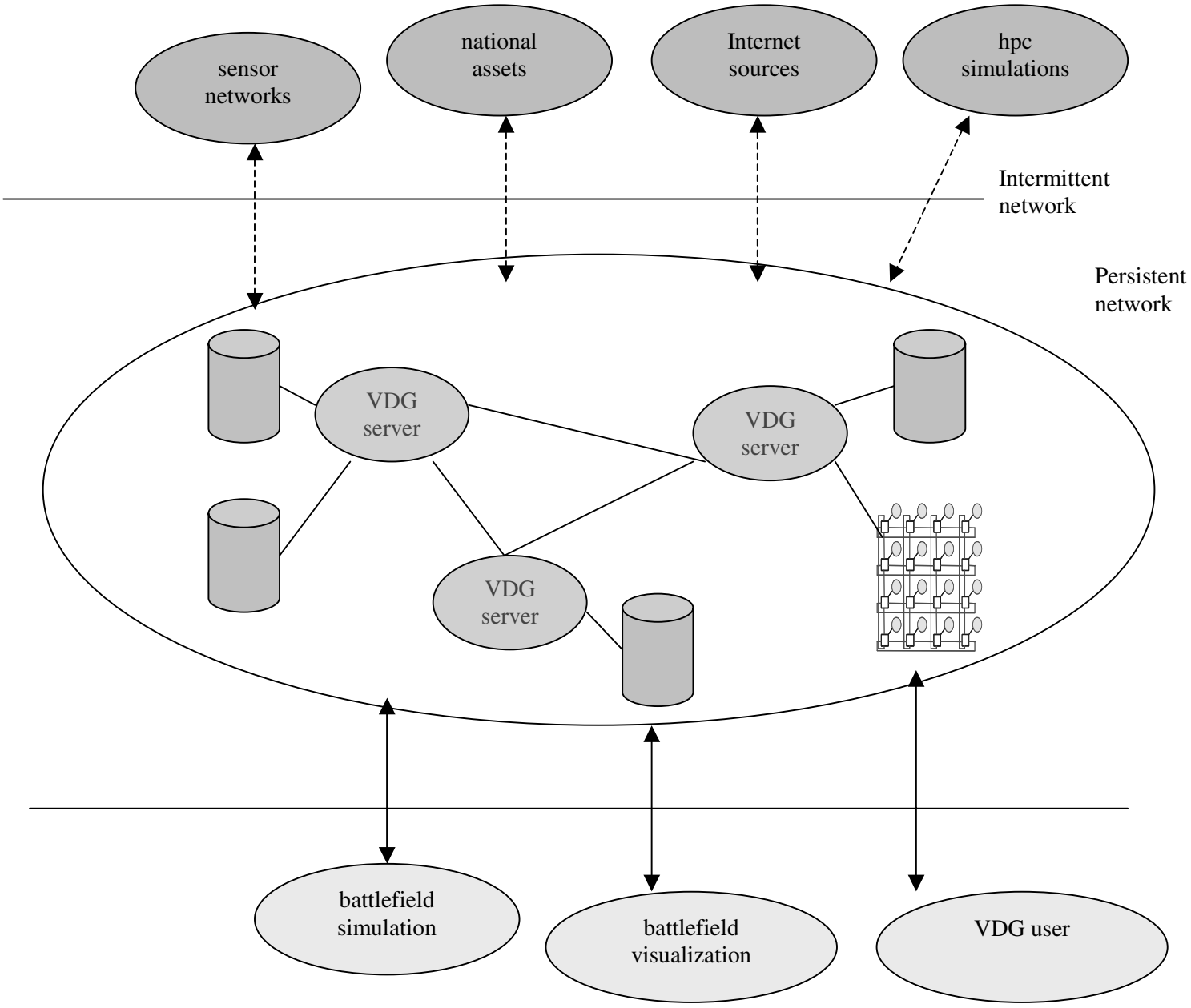


This kind of data grid could also support applications like battlefield simulation and visualization. This application would use these data sources coupled with up-to-date weather, terrain, and troop position data, to simulate complex and realistic scenarios. In addition, “what if” analysis may require on-demand launch of high performance scientific simulations to produce data needed within the simulation as well. This must also be accommodated in the VDG architecture.

3.0 System Architecture

The VDG system architecture is depicted below. It contains of a web of interconnected VDG servers each attached to large storage *depots*. Data producers (above) generate data that is stored to the VDG via a data producing API. Likewise, data consumers (below) consume data via an

API for data access. VDG servers provide a variety of services including data publishing, subscription, location, and discovery. In addition, VDG servers will move, replicate, cache, and stripe data objects for increased performance. Data objects are represented by meta-data (via XML) that describe the properties of the data, how to access it, how to find it, and possibly how to compute it. In the latter case, a data object might require the launching of computation to produce it. All data object representations are stored in a distributed registry that stores the location of all data objects. We are currently designing and building a VDG prototype at the AHPCRC using the Globus toolkit to provide low-level enabling infrastructure to move data efficiently and securely within the network, and to launch computations on remote resources.



4.0 Data Scheduling

To date, we have focused on one aspect of the VDG – date scheduling to achieve high performance. We describe our approach to this complex problem and some ideas we are currently exploring. Data scheduling must take into account the following:

- VDG server resources,

- VDG network bandwidth,

- VDG disk storage space,

- Client performance criteria: how soon must the data object be delivered to the client?

- Requested data size: how big is the size of the data object requested? This will determine the way in which it is delivered to the clients.

Data objects have a home location (or depot) in the VDG. Data objects will be moved (cached) at locations (depots) closer to the clients that use them. Therefore, caching strategies (replacement and validation) must be adopted When the cache is full, one or more cached objects has to be evicted to make space for the new incoming cached object through cache spillage. The replacement policy deals with cache spillage. A good replacement policy is required to select a cached object for removal that will least impact performance. The object selected for removal is ideally one that will not be used until the distant future.

The best performance is achieved by policies take into account the size of the objects evicted. LRU-MIN is a policy derived from LRU (least recently used) that tries to minimize the number of objects evicted by applying LRU only to the objects whose size is above some threshold. The threshold is adaptive: if not enough space has been released, the threshold is lowered and the policy is applied again. This policy outperforms every other removal policy. Another policy that gives a lower but comparable performance is LFU (least referenced file removed first). Most policies are some form of LRU. The advantage of LRU is its simplicity. Its drawback is it does not consider file sizes and latency. Before applying a cache replacement policy, the following factors should be considered:

- Inter-access time. Objects that have a smaller inter-access time are more likely to be requested in the future. Using this technique, the LRU algorithm will select the object with the largest inter-access time to be evicted.

- Number of Previous Accesses. Using the number of previous accesses made to an object is a good indication as to whether the object will be requested in future. This cannot be used alone as the deciding factor as it does not allow aging of a object, i.e. an object which has been accessed many times in the past but has not been accessed in a while.

- Object size. This is another important factor to be considered while caching. In caching, objects will be of different sizes. A high hit ratio may mean that more small objects should be cached. As more objects are found in the cache the higher will be the hit ratio.

- Cost for acquiring the object. It is important also to take into consideration the cost incurred in acquiring the object. The more expensive the object, the better it is to retain the object in the

cache. The object could be expensive because the network bandwidth to the originating server is small hence requiring more time to download.

Caching Strategies

We now present some caching strategies that we are exploring. Since the best strategy depends on the specific VDG and the kind of workload, we plan to implement a set of strategies and have the VDG learn which one is the most appropriate over time. Self-adaptive mechanisms for cache policies will be based on the hit-rates of the cache and thresholds. When the hit-rate falls below a specified threshold, a different caching policy will be selected automatically.

- Least-Recently-Used (LRU): evicts the object that was requested the least recently.

- Least-Frequently-Used (LFU): evicts the object that is accessed least frequently.

- Size: evicts the largest object.

- LRU-Threshold: is the same as LRU, except objects larger than a certain threshold size are never cached

- Hyper: is a refinement of LFU with last access time and size considerations included.

- Lowest-Latency-First: tries to minimize average latency by removing the object with the lowest download latency first.

- Hybrid: is aimed at reducing the total latency. A function is computed for each object that captures the utility of retaining a given object in the cache. The object with the smallest function value is then evicted. The function for a object p located at server s depends on the following parameters: c_s, the time to connect with server s, b_s the bandwidth to server s, n_p the number of times p has been requested since it was brought into the cache, and z_p, the size (in bytes) of object p. The function is defined as:

$$\frac{(C_s + W_b/b_s)\,(n_p)\,W_n}{Z_p}$$

where W_b and W_n are constants. Estimates for c_s and b_s are based on the times to fetch objects from servers in the recent past.

- Lowest Relative Value (LRV): includes the cost and size of an object in the calculation of a utility value for keeping an object in the cache. The algorithm evicts the object with the lowest value. The calculation of the value is based on extensive empirical analysis of trace data. For a given i, let P_i denote the probability that an object is requested $i + 1$ times given that it is requested i times. P_i is estimated in an online manner by taking the ratio D_i+1/D_i, where D_i is the total number of objects seen so far which have been requested at least i times in the trace. $P_i(s)$ is the same as P_i except the value is determined by restricting the count only to objects of size s. Furthermore, let $1- D(t)$ be the probability that a object is requested again as a function of the time (in seconds) since its last request t; $D(t)$ is estimated as:

$$D(t) = 0.035 \log(t + 1) + 0.45\,(1 - e^{-t/2e\sigma})$$

Then for a particular object *d* of size *s* and cost *c*, if the last request to *d* is the *i'th* request to it, and the last request was made *t* seconds ago, *d's* value in LRV is calculated as:

$$V(i, t, s) = \begin{cases} P_i(s)\,(1 - D(t))\ c/s, & \text{if } i = 1 \\ P_i\,(1 - D(t))\ c/s, & \text{otherwise} \end{cases}$$

Among all objects, LRV evicts the one with the lowest value. Thus, LRV takes into account locality, cost and size of an object.

- Greedy Dual-Size: This algorithm works as follows:

It associates a value, *H*, with each cached object *p*. Initially, when a object is brought into cache, *H* is set to be the cost of bringing the object into the cache (the cost is always non-negative). When a replacement needs to be made, the object with the lowest *H* value, *minH*, is replaced, and then all objects reduce their *H* values by *minH*. If an object is accessed, its *H* value is restored to the cost of bringing it into the cache. Thus, the *H* values of recently accessed objects retain a larger portion of the original cost than those of objects that have not been accessed for a long time. By reducing the *H* values as time goes on and restoring them upon access, the algorithm integrates the locality and cost concerns in a seamless fashion.

To handle different object sizes, we extend it by setting *H* to *cost/size* upon an access to an object, where cost is the cost of bringing the object into the cache, and size is the size of the object in bytes. We called the extended version the GreedyDual-Size algorithm. The definition of cost depends on the goal of the replacement algorithm: cost is set to 1 if the goal is to maximize hit ratio, it is set to the downloading latency if the goal is to minimize average latency, and it is set to the network cost if the goal is to minimize the total cost.

At the first glance, GreedyDual-Size would require *k* subtractions when a replacement is made, where *k* is the number of objects in cache. However, a different way of recording *H* removes these subtractions. The idea is to keep an "inflation" value *L*, and let all future setting of *H* be offset by *L*. Below is an efficient implementation of the algorithm.

Initialize L = 0. Process each request object in turn: The current request is for object p:

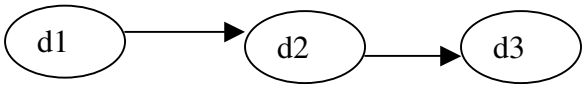
1. if p is already in cache,
2. H(p) = L + c(p)/s(p).
3. if p is not in cache,
4. while there is not enough room in cache for p,
5. Let $L = min_{q \in M} H(q)$
6. Evict q such that H(q) = L.
7. Bring p into cache and set H(p) = L + c(p)/s(p)
end

Using this technique, GreedyDual-Size can be implemented by maintaining a priority queue on the objects, based on their *H* value. Handling a hit requires *O(log k)* time and handling an eviction requires *O(log k)* time, since in both cases a single item in the queue must be updated. More efficient implementations can be designed that make the common case of handling a hit requiring only *O(1)* time.

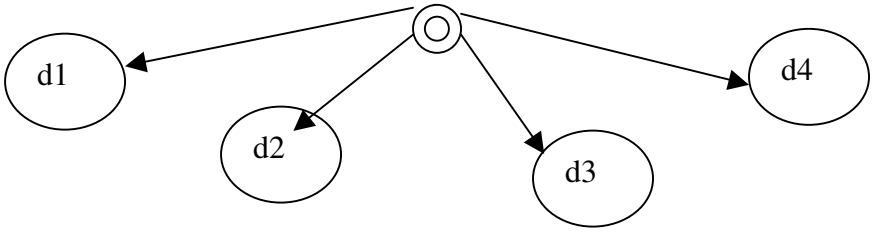
Pre-Staging Strategies: When to Cache?

The decision to cache an object can be made when the object is accessed or by pre-staging objects. The latter can be achieved with knowledge of prior data access patterns. Pre-staging is crucial to reducing object access latency when performance demands are tight. Such access patterns can be explicitly provided by the client (e.g. "I will want an update of the satellite weather data every hour") or in future work, inferred by the system. In addition to timing constraints for single object access such as above, there are data object access patterns to consider when multiple data objects are accessed by the client:

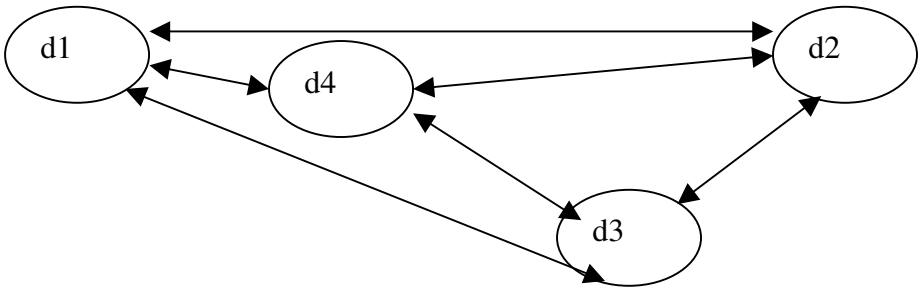
- sequential: Client accesses data objects d1, then d2 and finally d3.



- concurrent: Client accesses data objects d1, d2, d3 and d4 concurrently.



- Out-of-Order: For example, client could accesses data objects d1, d2, d3 and d4 in any order.



We are currently investigating how to pre-stage data objects given both temporal and spatial access patterns. For example, in concurrent access, ideally all of the data objects should be pre-staged locally simultaneously. On the other hand, in sequential access, there is more flexibility: d1 should be staged locally first, and the d2, and so on.

6.0 Related Work

A number of research groups have implemented infrastructures similar to the VDG but differ in the applications supported or in the underlying data scheduling support. OceanStore [OS] is a utility infrastructure designed to span the globe and provide continuous access to persistent information. It consists of untrusted servers and data is protected through redundancy and cryptographic techniques. The concept of "data cached anywhere, anytime" is proposed. VDG differs itself from OceanStore as it also determines the best data objects to cache and where and when to do so. The Internet Backplane Protocol (IBP) [NS] controls storage that is implemented

as part of the network fabric itself. It allows an application to control intermediate data staging operations as data is communicated between processes. The application can exploit locality and manage scarce buffer resources effectively. VDG differs from IBP by making more precise decisions on data locality and scarce resource management through identification of common data object access patterns, but it shares the concept of storage depots. Globus Data Grid is an infrastructure [GL] with services for storage, metadata, replica and cache management, replica creation and selection. VDG differs itself from the Globus Data Grid by implementing scheduling policies for data objects based on cache replacement, data access patterns, object popularity, and network resources.

Acknowledgements

This work was sponsored in part by the Army High Performance Computing Research Center (AHPCRC) under the auspices of the Department of the Army, Army Research Laboratory cooperative agreement number DAAD19-01-2-0014.

7.0 Bibliography

[NS] J.S. Plank et al., "The Internet Backplane Protocol: Storage in the Network," *Proceedings of NetStore* 1999.

[OS] J. Kubiatowicz et al., "OceanStore: An Architecture for Global-Scale Persistent Storage," *Proceedings of the Ninth international Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 2000),* November 2000.

[GL] Globus: `www.globus.org`, 2002.